Evaluation of an FPGA-Based Shortest-Path-Search Accelerator

Yasuhiro Takei, Masanori Hariyama and Michitaka Kameyama
Graduate School of Information Sciences, Tohoku University
Aoba 6-6-05, Aramaki, Aoba, Sendai, Miyagi, 980-8579, Japan
Email: {takei, hariyama, kameyama}@ecei.tohoku.ac.jp

Abstract— *Shortest-path search over large scale graphs is widely used in various applications. However, shortest path algorithms such as Dijkstra's algorithm include complex processing. It is difficult for accelerators such as GPUs to accelerate these algorithms efficiently. This paper presents an FPGA-based accelerator with SIMD architecture for the shortest-paths algorithm. In the proposed architecture, processing of the Dijkstra's algorithm is done with a high degree of parallelism, and the memory usage is reduced by the replacement of the node data. According to the evaluation, the processing time of the proposed architecture is about a half of that of a CPU, and the amount of the node data stored in on-chip memory is about one-third of all nodes when the input graph is a lattice graph.*

Keywords: Shortest-path search, Dijkstra's algorithm, Single instruction multiple data (SIMD), FPGA

1. Introduction

Recently, there is a huge demand of processing large-scale graphs. Especially, finding the shortest-path in large scale graphs is used in many applications such as traffic simulation, social networking service and bioinformatics. To solve the shortest-path problem, various algorithms has been proposed. Dijkstra's algorithm [1] and Bellman-Ford algorithm [2] were proposed to solve the single-source shortest-path problem (SSSP). Warshall-Floyd Algorithm [3] was proposed to solve the all-pair shortest-path problem (APSP).

To accelerate processing speed of solving shortest-path problem, there have been many software-based studies in terms of improving a data structure and reducing a computational amount. In order to process the shortest-path problem for large scale graphs, PC clusters with many CPUs are often used [4] because of their large memory capacity. However, these computing systems need very large space and power consumption.

Some studies used GPUs for solving shortest-path problem. Harish [5] and Katz [6] have implemented shortest-path search on the GPU. GPUs are suitable for simple and parallelized processing. However, it is difficult to accelerate shortest-path searching efficiently when the shortest-path algorithm includes serial and complex data-flows.

Other studies used the FPGA-based accelerator for solving shortest-path problem. FPGAs can implement application-specific data-paths by reconfiguration after fabrication. Moreover, the power consumption of FPGAs are less than one-tenth of that of CPUs and GPUs. Tommiska [7], Fernandez [8], and Sridharan [9] have designed the FPGA-based architecture for SSSP with the Dijkstra's algorithm. Bondhugula [10] has designed the FPGA-based architecture for APSP with the Warshall-Floyd algorithm. However, their works did not consider processing large-scale graphs since the memory usage of the input graph is not considered.

To solve these problems, we design an FPGA-based accelerator for the Dijkstra's algorithm on large scale graphs. In order to accelerate processing and memory access, we design the SIMD (single instruction multiple data) architecture. We explain how to search the shortest path with a high degree of parallelism, and how to replace the node data on a limited memory space. In this paper, we implement the improved architecture from our previous work [11], and we evaluate the memory usage and processing time of the shortest path search.

2. Dijkstra's algorithm and its implementation on an FPGA

The Dijkstra's algorithm is one of the most popular algorithms to solve SSSP. Because it is easy to implement, this algorithm is used in various applications such as analysis of the internet, traffic simulation and so on. Let S be the node where we are starting. Let

$d(y)$ be the distance from S to node y. The flow of the Dijkstra's algorithm is represented by the following steps.

Step1: Assign to every node a tentative distance: set it to zero for S, and to infinity for all other nodes. Mark all nodes *"unvisited"*.

Step2: Select the *unvisited* node which has the smallest tentative distance and make it the *"current node"*.

Step3: For the *current node*, consider all of *unvisited* neighbor nodes and update their tentative distance. If the *current node* is A , and one of the *unvisited* neighbor node is B, set the tentative distance of B ($td(B)$) to $\min(td(B), d(A) + l_{AB})$,where l_{AB} is the length of the edge between A and B. When considering all of *unvisited* neighbor nodes of the *current node*, mark the *current node "visited"*.

Step4: Until all nodes are marked *visited*, go back to Step2.

The processing time of the Dijkstra's algorithm depends on searching the minimum distance in Step2 and updating tentative distances in Step3. In these processing, there are many comparison operations on multiple node data. Hence, a parallelized architecture such as the SIMD architecture is suitable for accelerating the processing of the Dijkstra's algorithm.

Since tentative distances and paths are read and updated frequently, on-chip memory on an FPGA is suitable for storing these data. However, the capacity of the on-chip memory is small. The memory management is required for reducing the on-chip memory usage and the total processing time. In the Dijkstra's algorithm, tentative distance of nodes that connects current or visited nodes is only used in the processing. As shown in Fig.1, the current node (C) and unvisited nodes connected to current or visited nodes (D,E) are only used in the processing until the next current node is determined. As shown in Fig.2, after the next current node (D) is determined, a previous current node (C) data is unnecessary in the processing. Hence the memory space for the previous current node data (C) can be reused for the new node data (F).

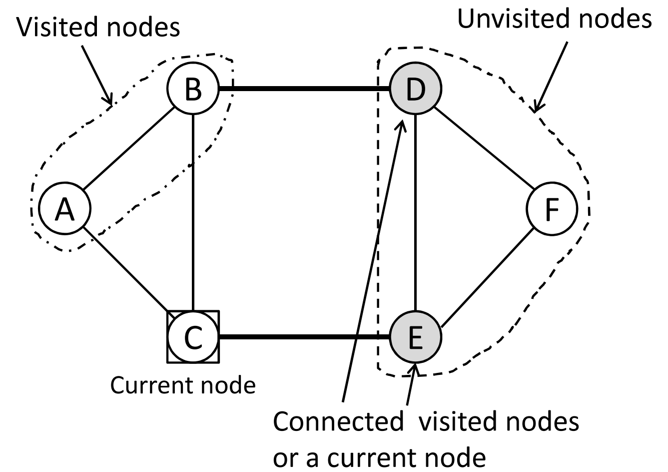

Fig. 1: The current node (C) and unvisited nodes connected to current or visited nodes (D,E)

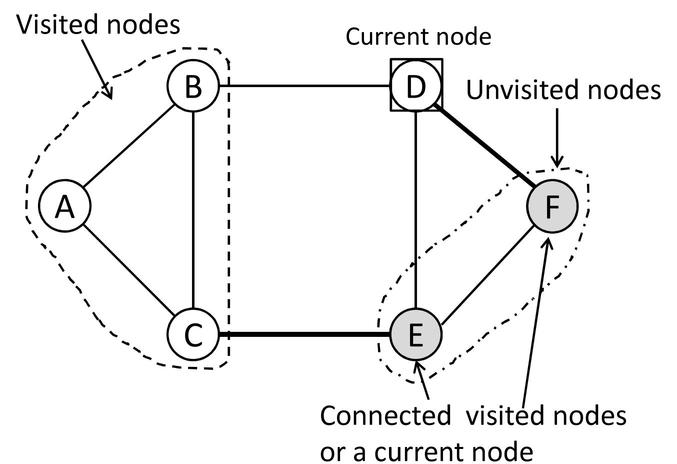

Fig. 2: The current node (D) and unvisited nodes connected to current or visited nodes (E,F)

3. Architecture

Figure 3 shows the overall architecture. This architecture consists of an external memory, a CPU core and a Dijkstra module. An external memory such as a DDR2 SDRAM stores the adjacency list of the input graph. The Dijkstra module consists of processing elements (PEs), selectors, a decoder, a current node register, and an address generation unit (AGU). The current node register stores the current node number and the distance from the start to the current node.

Figure 4 shows the architecture of the processing element in the Dijkstra module. This architecture consists of a node memory, modules for searching for

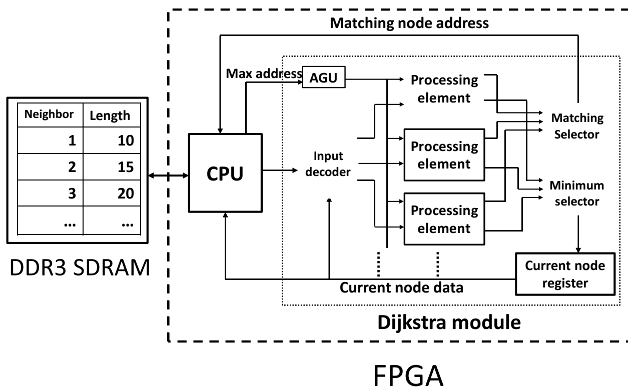

Fig. 3: Overall architecture

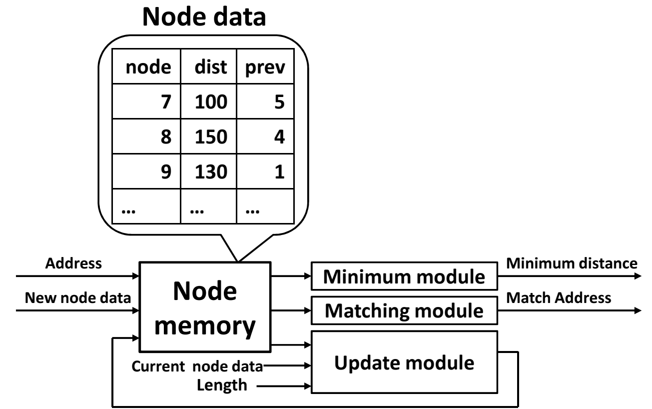

Fig. 4: Architecture of the PE

minimum distance, for matching the node number and for updating the tentative distance. These modules consist of comparators, registers and adders. The node memory stores the values of node number, the tentative distance and the previous node number of the shortest path.

For updating the tentative distances in node memories, the neighbor node number is searched by matching node modules in parallel as shown in Fig.5. In this case, the node number 8 is searched for. Then the tentative distance at node 8 is compared with the sum of the distance at the current node 6 and the length of the edge 6 to 8 by the update module. If the sum of the distance at the current node and the length the edge is smaller than the tentative distance, the tentative distance and the previous node number are updated as shown in Fig.6.

For the searching for the minimum distance in node memories, minimum distance modules and minimum selector are connected as shown in Fig.7, and searching in parallel. In this case, node 7 has the minimum distance, and node 7 is selected as the new current distance. When the minimum distance searching is completed, the new minimum distance and the node number are stored in the current node register.

After these data are stored in the current node register, the memory space for the current node can be overwritten to the first unvisited neighbor node data that have not existed in node memories as shown in Fig.8. If the number of these new neighbor nodes is more than two, the empty space in node memories is used for storing subsequent unvisited neighbor node data.

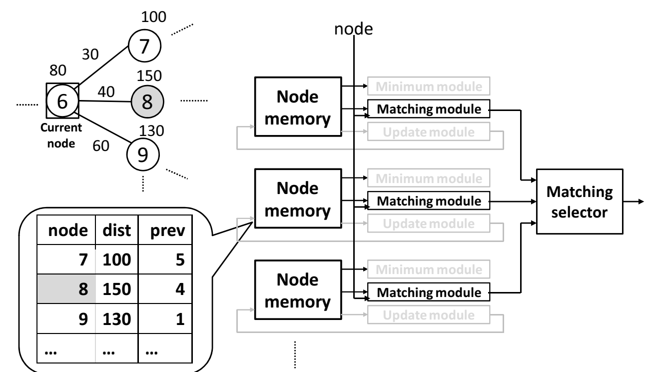

Fig. 5: Searching a node number data in node memories

4. Evaluation of the proposed architecture

In this evaluation, we use the Terasic DE4 FPGA board [12]. This board includes an Altera Stratix IV GX EP4SGX530, and a DDR2 SDRAM (4GB). Altera Quartus 13.1 is used for the FPGA implementation. For a proto-type design, we implement the proposed architecture with 8 PEs. 4,096 nodes can be processed in the implemented architecture. NiosII soft-core processor [13] is used for the CPU core as shown in Fig.3. It is designed using Altera Qsys 13.1 and programmed by C language using Nios II EDS 13.1. Table 1 shows the resource usage of the proposed accelerator. The usage of memory bits of NiosII is larger than that of the Dijkstra module since the programming code for storing the graph is large. Considering with the resource usage of the on-chip memory, about 300,000 nodes can be implemented on the FPGA board if the programming code on the NiosII is improved.

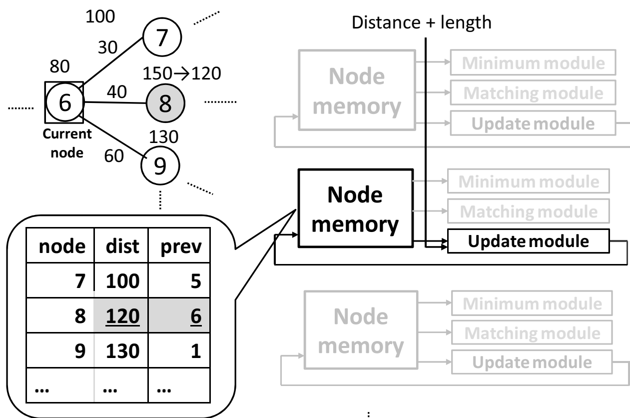

Fig. 6: Updating node data in the node memory

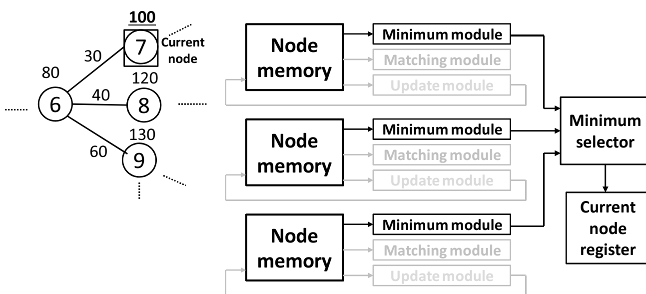

Fig. 7: Searching the minimum distance in node memories

Let us compare the performance of the FPGA-based accelerator with that of Intel Core2 Quad processor. We implement the shortest-path problem on a lattice graph as shown in Fig.9. The Dijkstra's algorithm is implemented on the Core2 Quad processor by using C++ language. Microsoft Visual studio 2010 is used for compiling. Table 2 shows the processing time comparison. The processing time of the proposed architecture is about half of the CPU. The performance of the proposed architecture can increase when more PEs are implemented on the FPGA. Moreover, FPGAs with hard-core CPUs, such as Xilinx Zynq [14] and Altera Cyclone V SoC [15] can be used in order to reduce the control overhead. These FPGAs includes multicore CPUs such as the ARM Cortex-A9. The performance of these CPU cores is more than ten times as much as that of the NiosII core.

Let us consider the memory usage of the node memories. Table 3 shows the number of node data in node memories in the processing of the Dijkstra's

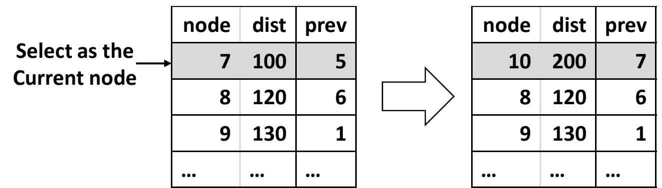

Fig. 8: Overwriting the current data to a new node data in the node memory

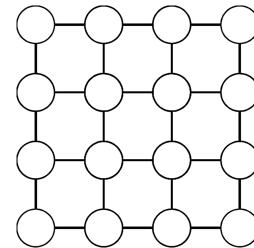

Fig. 9: Lattice graph

algorithm. According to the ratio of nodes in the node memories to all nodes in the graph, the memory usage is about one-third of the all nodes data when the input graph is a lattice graph. As a result, about three times nodes of the node memories can be processed on the FPGA board when the input graph is as sparse as the lattice graph, such as a map data. Memory usage depends on a structure of the input graph. When the input graph is sparse, usage of the node memories becomes small. On the other hand, usage of the node memories becomes at maximum when the start node connects to all other nodes. In this case, (All nodes - 1) nodes are stored in the node memories.

5. Conclusions

We have proposed an FPGA-based accelerator for shortest-path search. We designed for processing the Dijkstra's algorithm in parallel and we explained the replacement of the node data to reduce the memory usage. According to the evaluation, the processing time of the proposed architecture is about half of the CPU, and the usage of the on-chip memory is about one-third of the all nodes when the input graph is a lattice graph.

The suitable architecture of the shortest-path-search accelerator depends on the structure of a input graph. The proposed architecture is suitable for sparse graphs. However, this architecture is not suitable for dense graphs since the usage of the node memories cannot reduce so mach. Hence, we should design another

Table 1: Resource usage

	LUT	Register	Memory bit	DSP
Dijkstra module	1509	607	196608	0
NiosII (CPU)	9298	12507	1832318	4

Table 2: Processing time(ms)

Input graph	FPGA (50MHz)	Core2 Quad (2.83GHz)
1024Nodes, 3968Edges	9.38	16.00
4096Nodes, 16130Edges	50.40	94.00

architecture for dense graphs, such as the GPU-like architecture for processing an adjacency matrix.

Moreover, to process a very large scale graph, it is important to reduce the bottleneck of the data-transfer of the graph data from the external storages such as SSDs to FPGA boards. We are going to implement the framework for graph compressing such as WebGraph [16] or Graphillion [17] on the FPGA board.

Acknowledgement

This work is supported by JSPS KAKENHI grant number 24300013 and Grant-in-Aid for JSPS Fellows grant number 15J04973.

Table 3: The number of node data in node memories

All nodes in a graph	256	1024	4096
Nodes data in node memories	87	316	1329
Ratio of the node data	0.340	0.309	0.324

References

[1] E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs", Numerische Mathematik, 1(1): pp.269–271, 1959.

[2] R. Bellman, "On a Routing Problem", Technical report, DTIC Document, 1956.

[3] R. W. Floyd, "Algorithm 97: Shortest Path", Commun. ACM, 5(6) pp.345–346, June 1962.

[4] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. "Pregel: a System for Large-Scale Graph Processing", In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, pp. 135–146, 2010.

[5] P. Harish, and P. J. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA", High performance computing-HiPC 2007, Springer, pp.197-208, 2007.

[6] G. J. Katz and J. T. Kider Jr, "All-Pairs Shortest-Paths for Large Graphs on the GPU", In Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware, 2008.

[7] M. Tommiska and J.Skytta. "Dijkstra's Shortest Paths Algorithm in Reconfigurable Hardware", In Proc. Field Programmable Logic and Applications, pp. 653–657, 2001.

[8] I. Fernandez, J. Castillo, C. Pedraza, C. Sanchez, and J. I. Martinez, "Parallel Implementation of the Shortest Path Algorithm on FPGA" In Proc. 4th Southern Conf. on Programmable Logic., pp. 245–248, 2008.

[9] K. S. T.K.Priya and P. Kumar, "Hardware Architecture for Finding Shortest Paths", In Proc. IEEE Region 10 Conf., pp. 1–5, 2009.

[10] U. Bondhugula, A. Devulapalli, J. Fernando, P. Wyckoff, and P. Sadayappan, "Parallel FPGA-Based All-Pairs Shortest-Paths in a Directed Graph", In Proceedings of Parallel and Distributed Processing Symposium, 2006.

[11] Y. Takei, M. Hariyama and M. Kameyama, "An SIMD Architecture for Shortest-Path Search and Its FPGA Implementation ", International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), pp.53–56, 2014

[12] Terasic, "Altera DE4 Development and Education Board", http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=EnglishNo=501.

[13] Altera, "Nios II Processor", http://www.altera.com/devices/processor/nios2/ni2-index.html.

[14] Xilinx, "Zynq-7000 All Programmable SoC", http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/index.htm.

[15] Altera, "Cyclone V SoCs: Lowest System Cost and Power", http://www.altera.com/devices/processor/ soc-fpga/cyclone-v-soc/cyclone-v-soc.html.

[16] P. Boldi and S. Vigna. ,"The Webgraph Framework I: Compression Techniques". In Proc. of the 13th international conference on World Wide Web, pp. 595–602. ACM, 2004.

[17] T. Inoue, H. Iwashita, J. Kawahara, and S. Minato. "Graphillion: Software Library for Very Large Sets of Labeled Graphs", International Journal on Software Tools for Technology Transfer, 2014.